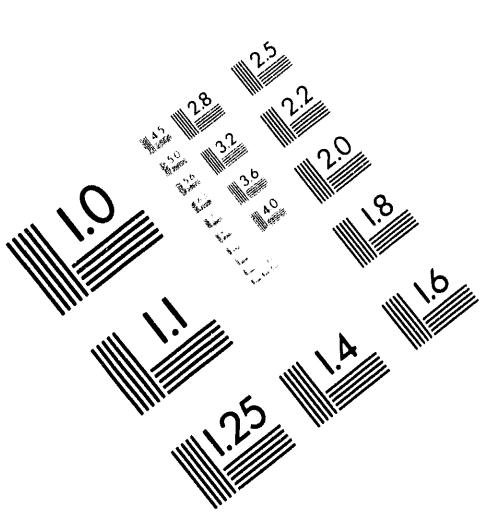
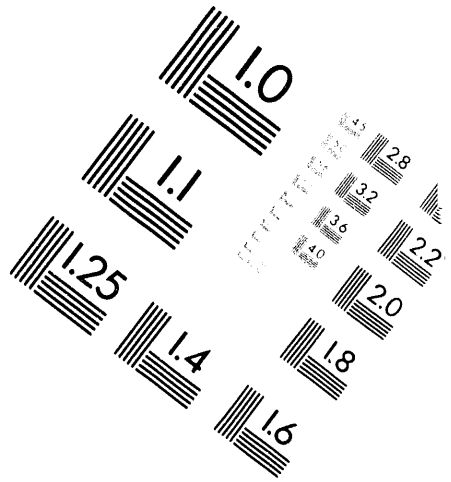
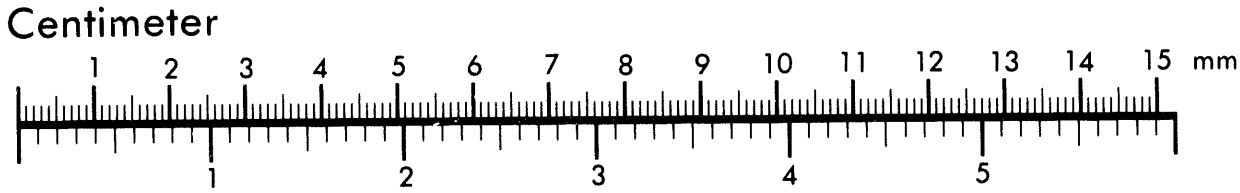


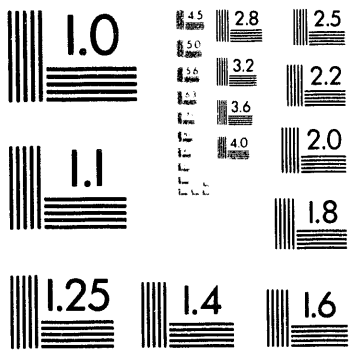
Association for Information and Image Management

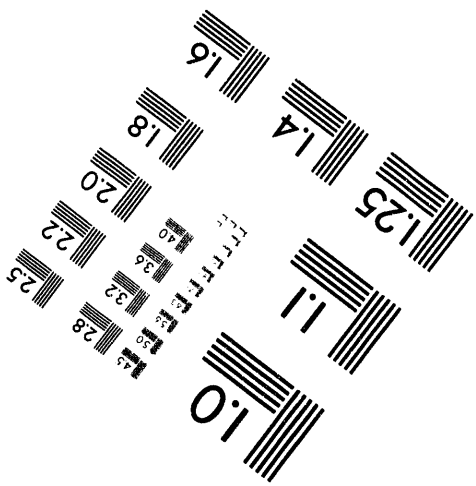
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

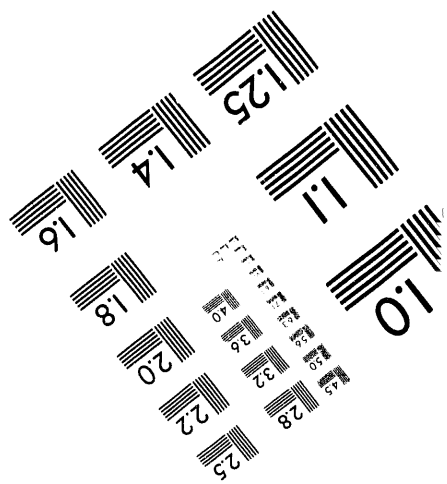
301/587-8202











MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

Active Messages Versus Explicit Message Passing Under SUNMOS*

Rolf Riesen
Sandia National Laboratories
Albuquerque, NM 87185-1109
rolf@cs.sandia.gov

Arthur B. Maccabe
University of New Mexico
Albuquerque, NM 87131
maccabe@cs.unm.edu

Stephen R. Wheat
Sandia National Laboratories
Albuquerque, NM 87185-1109
srwheat@cs.sandia.gov

Abstract

In the past few years much effort has been devoted to finding faster and more convenient ways to exchange data between nodes of massively parallel distributed memory machines. One such approach, taken by Thorsten von Eicken et al.[6] is called Active Messages. The idea is to hide message passing latency and continue to compute while data is being sent and delivered.

We have implemented Active Messages under SUNMOS for the Intel Paragon and performed various experiments to determine their efficiency and utility. In this paper we concentrate on the subset of the Active Message layer that is used by our implementation of the Split-C library. We compare performance to explicit message passing under SUNMOS and explore new ways to support Split-C without Active Messages. We also compare our implementation to the original one on the Thinking Machines CM-5 and try to determine what the effects of low latency and low bandwidth versus high latency and high bandwidth are on user codes.

1 Introduction

Over the past few years, network bandwidth has increased dramatically, while latency, the time it takes to access the network, has decreased. This is true for distributed environments using ATM and FDDI, as well as massively parallel systems such as the Intel Paragon with a peak bandwidth of 200 MB/s in fast streaming mode.

Unfortunately, software has not kept up with this trend. This is evident in OSF/1 AD for the Intel Paragon. While the hardware is capable of achieving 175 MB/s in slow streaming mode, a user level code gets at most 65 MB/s for large messages. User level latencies are on the order of 45 micro seconds[4].

Some researchers try to hide this latency and perform other computations while data is in transit. Others try to minimize system software overhead, and yet others try to accomplish both. Split-C and active messages have received much attention because of their attempt to tackle the problem as a whole and provide a novel form of message passing.

We have implemented active messages and ported Split-C to SUNMOS on the Intel Paragon. In this paper we compare the performance of explicit message passing, active messages, and Puma portals.

In section 2 we give an overview of active messages and describe how we implemented them in SUNMOS. Section 3 gives a brief introduction to the two types of Puma portals [7] used for the measurements in this paper. In Section 4 we look at Split-C, our implementation and its relationship to active messages and Puma portals. We are now ready to run test codes and measure the performance of the various paradigms. The results are presented in section 5. The paper concludes with a comparison of Split-C and active messages implementations on the Thinking Machines CM-5 and the Intel Paragon.

2 Active Messages

Active messages have been proposed in [6] and are the lowest message passing level available on the CM-5[5]. Other communication paradigms on the CM-5 are implemented using active messages.

An active message consists of a function address and four[1] 32 bit parameters (20 bytes). Since all the nodes in a SPMD application share the same memory map, it is possible for a node to determine the start

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

[1]The latest implementation allows up to 17 parameters (72 bytes)[5].

address of a function on another node by looking at the same function on the local node.

When an active message arrives, program execution continues at the address specified in the active message header. This function, called an active message handler, is invoked with the parameters supplied by the sender. When the active message handler finishes, program control returns to the point just prior to the active message arrival.

There are two reasons why this method is fast on the CM-5 and can be used as the basic message passing mechanism. First, in order to send an active message no access or alignment checks are needed, since the active message send function is called with value parameters only. The user cannot send data that is not in her address space. The function address does not need to be verified either. If control is transferred to a non-existing function or outside the user's address space, the memory management unit of the receiving node will fault the user process.

The second reason for the efficiency of active messages on the CM-5 is the fact that the communication hardware is accessible from user level. Inserting and extracting messages into and from the communication network can be done without trapping into the kernel. This also means, that the active message layer on the CM-5 has to poll the network for new messages. While this makes receipt and dispatch of active messages quick, it can cause delays, if the receiving node is very busy.

The Intel Paragon does not provide direct user access to the communication hardware. The SUNMOS implementation of active messages therefore requires a trap into the kernel to send, and an interrupt on the receiving end to receive an active message. Because of this, active message startup and dispatch times are much higher under SUNMOS than they are on the CM-5. However, incoming requests are handled as they arrive without the need for application level polling. Thus, incoming requests are not delayed while the application is involved in a lengthy computation.

To compare the relative merits of the approaches we coded a simple matrix multiply routine. Under SUNMOS using Split-C and active messages this code ran much faster than the same code under an OSF implementation of Split-C that uses polling. The SUNMOS implementation also outperformed the original CM-5 implementation by a factor of seven [3] [1].

3 Puma Portals

Puma, the successor to SUNMOS, currently under development at Sandia National Laboratories, offers portals as the basic message passing mechanism. Portals were designed to be very efficient and simple, yet powerful enough to support any of the currently available and proposed message passing paradigms[7], [2]. In this paper we concentrate on two of the four portal types provided by Puma.

A *readmem* portal allows a process on a node to specify a region of its memory space to be readable by other processes in its group. Once a memory region has been declared to be readable, processes on other nodes can issue a request that are handled by the kernel on the node with the readmem portal. Therefore, once a readmem portal has been setup, no context switch to the user level is required to read data from the readmem portal region.

A *writemem* portal is used by a process to make parts of its memory writable by other processes in its group. When a kernel receives data to be put into a writemem portal, it certifies the validity of the request and the DMAs the data directly into the desired memory region. Again, no context switch to the user level is required for the operation to complete.

Puma is not yet available on the Intel Paragon. For this paper we have changed the SUNMOS kernel to do the necessary processing for readmem and writemem portals. The numbers reported are therefore very close to the numbers we expect for the Puma kernel later this year.

4 Split-C

Split-C is built on top of active messages. It is a superset of the C programming language and offers a two dimensional view of a global shared memory. In contrast to high performance Fortran and other similar approaches, it offers the programmer a clear cost model for memory access. The programmer decides the data distribution and is always aware when a data access involves a remote node. Split-C distinguishes between local and global pointers. Local pointers can only point to memory locations on the local node. A global pointer consists of an address and a node number to specify a coordinate in the two dimensional memory space[1].

Split-C gets its name from the possibility to split the data access request and actual delivery into two parts; the so called "split-phase" read operation. In between the programmer can use the time to continue

```
double *global gPtr;
double local, x;

/* Split phase assignment */
local:= *gPtr;

/*
** other processing... state of
** "local" unknown.
*/

sync();

/*
** The global value pointed to by
** "gPtr" is now available in the
** variable "local".
*/

x = local * 3.0;
```

Figure 1: Split-C Code Fragment

computation. The example in Figure 1 should clear this up:

The assignment to the variable `local` will cause a data transfer from another node[2]. Once the request has been issued, computation can continue. Before the value of `local` is used, a call to `sync()` is on order. The call will block until the data has arrived, or return immediately, if it is already there. Note that `sync()` is not a global operation. Only the node issuing the `sync()` is blocked.

The Split-C team at Berkeley has modified the gcc compiler and ported it to various architectures. We have retargeted the compiler to run on Sun workstations and produce code for the Paragon. We also rewrote the Split-C library to take advantage of specific SUNMOS features. Doing this showed that Split-C can be implemented without active messages. While active messages are more general than readmem and writemem portals, the latter is all that Split-C needs. We have plans to port Split-C to Puma once it becomes available. In the meantime, the Split-C library uses active messages under SUNMOS as well as explicit message passing for larger data transfers.

[2]This assumes `gPtr` points to another node. Local access through global pointers is possible, but less efficient than access though a local pointer.

5 Data Transfers

In this section we measure the time it takes to write and read data from a remote node using explicit message passing, active messages, and portals. In Split-C, an assignment of a single double precision floating point number to a global location causes the transfer of eight bytes from one node to another. Of course, it is possible to group several small requests together and transfer all the data in a single operation. While it makes sense to do that, even for explicit message passing codes, the difference in transmit time between the three methods under investigation is diminishing for large messages.

Table 1 illustrates this. The first column shows the size of remote read requests in kilo bytes. The second, third, and fourth column show the time in micro seconds it takes to complete the request using active messages, explicit message passing, and readmem portals respectively. While there is a measurable difference between the three methods, its effect becomes less and less important. This is shown in the last two columns of the table. We assume the readmem portal time for a given size to be 100%. The second to last column then shows how much more, in percent, it takes to complete the transfer using active messages. The last column shows the difference in percent between readmem portals and explicit message passing. For this reason we only consider data transfers of less than 256 bytes for the reminder of this paper.

5.1 Remote Read Using Active Messages

The algorithm used to read memory on a remote node, using active messages, is a follows:

1. The local node sends an active message to the remote node. The message contains the address of a simple active message handler, the amount of requested data, and the start address of the data.

2. The message handler invoked on the remote node simply uses the length and address information to send a message containing the requested data back to the local node.

3. The local node has preposted a receive and the data will be deposited in the local user memory.

Figure 2 shows the result for all three methods for message sizes from 4 to 256 bytes. Each data point is the result of 100 trials. The plot shows the minimum, average, and maximum observed for each size measured.

Size in k Bytes	AM	msg	portal	AM vs. portal	msg vs. portal
1	160.0μs	146.1μs	118.0μs	35.6%	23.8%
5	181.3μs	166.8μs	139.0μs	30.4%	20.0%
10	216.5μs	204.0μs	174.8μs	23.9%	16.7%
50	471.2μs	457.1μs	428.7μs	9.9%	6.6%
100	809.9μs	793.1μs	765.8μs	5.8%	3.6%
500	3366.9μs	3345.6μs	3319.7μs	1.4%	0.8%
1000	6549.9μs	6521.5μs	6499.6μs	0.8%	0.3%

Table 1: Large Data Transfers (Read)

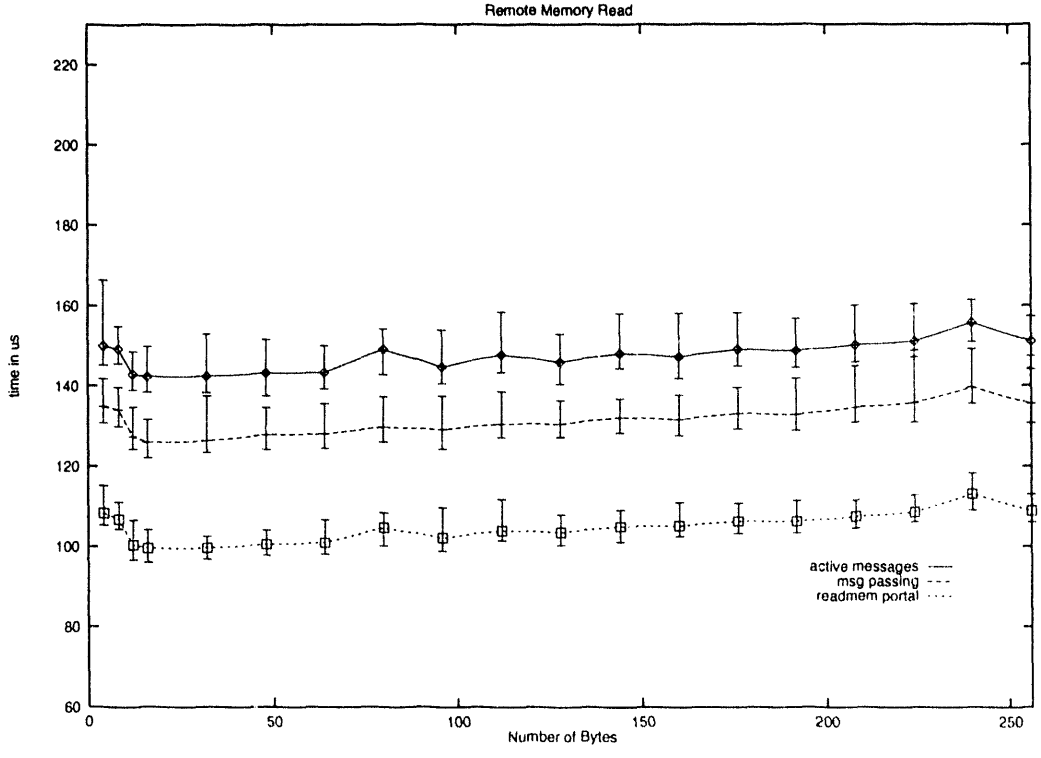


Figure 2: Remote Read Access

5.2 Remote Read Using Portals

The steps involved to read remote memory using a readmem portal are as follows:

1. The local node preposts a receive for the requested data and then sends a request to the remote readmem portal. The request contains the address and amount of data to be read.
2. The kernel on the remote node handles the request and sends the desired data.
3. The data arrives at the local node and the receiver is informed through a flag that the data has arrived.

5.3 Remote Read Using Explicit Message Passing

For comparison reasons we include the measurements for a explicit message passing remote read. It works as follows:

1. The remote node posts a receive for readmem requests.
2. The local node sends a request containing the address and amount of data requested.
3. The remote node receives the request, handles it, and sends the desired data to the local node.
4. The local node has preposted a receive for the data. It will be deposited in the local user memory.

Note that the semantics of this algorithm are slightly different than that of the previous two. The user code on the remote end has to anticipate the time when the readmem request arrives and prepare accordingly. If the code is not ready for the request, or it is busy doing other things, the data transfer will be delayed. Portals and active messages do not have this problem. As soon as a request arrives, it will be handled, no matter where local program control resides at that instant. We have included explicit message passing here only to compare transmission times. We do not imply that Split-C could be implemented efficiently using explicit messages passing only.

Figure 2 clearly illustrates that using portals under SUNMOS is superior to active messages and explicit message passing. In both cases that has to do with the time that is saved by handling the request in the kernel versus a context switch into user space. Our implementation of active messages makes the problem even worse, since two context switches are required. One to activate the message handler thread, and a second one to return to the regular user thread.

5.4 Remote Write Using Active Messages

In order to measure writemem performance using active messages, we proceed as follows:

1. The local node sends an active message containing the amount of data to be shipped, as well as the address on the remote node where it should be deposited.
2. Immediately after that the local node sends a message containing the write data.
3. The active message handler on the remote node issues a receive into the location specified by the parameters sent in the active message.

At this time the operation is complete. To get accurate timings, we have the remote node send the data back in the same fashion. We measure the total time and then divide by two. Figure 3 shows the results.

5.5 Remote Write Using Portals

To send data to a writemem portal the local node simply sends the data to the portal on the remote node. The kernel will receive the request and DMA the data into the user memory. In order to measure the time it took, the test code on the remote node polls the memory location to detect when the data arrives. It then turns around and sends the data back, so the local node can measure the total time and divide by two.

5.6 Remote Write Using Explicit Message Passing

Again, we include this test for comparison reasons only. Since the remote node has to be ready and poll for incoming writemem requests, we do not feel that explicit message passing can be substituted for portals or active messages.

Our test code posts a receive for the writemem request on the remote node. The local node sends a first message containing the address and the amount of data. A second message sent immediately afterwards contains the data. Once the remote node receives the first message, it can issue a receive request for the desired amount into the appropriate memory location.

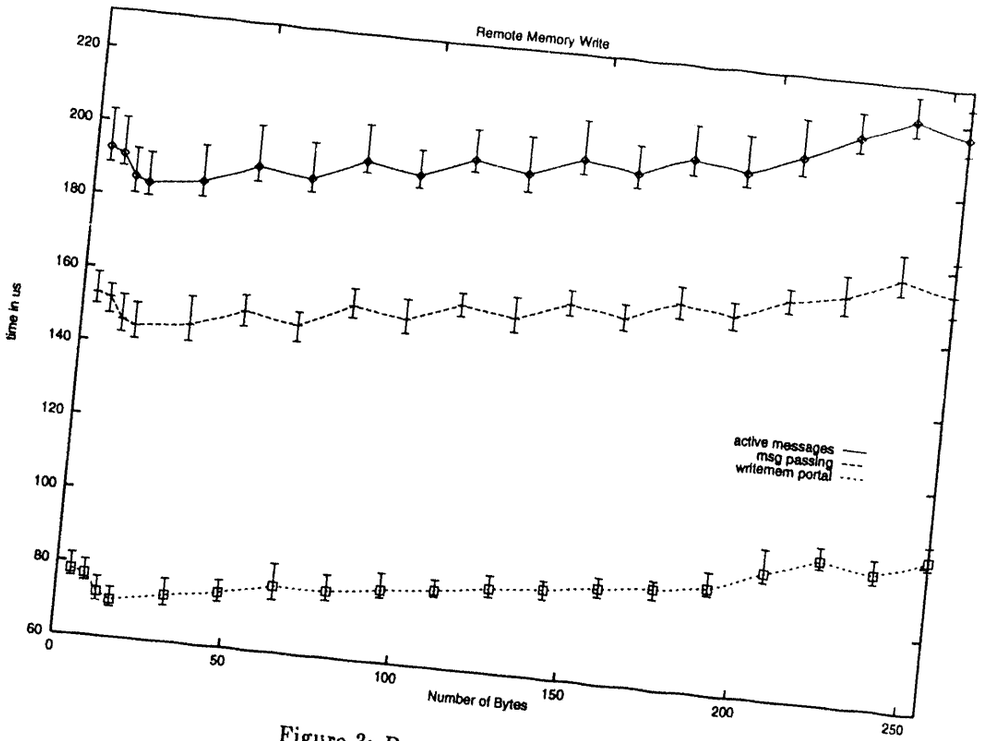


Figure 3: Remote Write Access

6 Summary

We have seen that Split-C can be implemented using portals or active messages as the lowest message passing mechanism. A comparison of remote data access using portals, active messages, and explicit message passing shows that portals have superior performance compared to the other two mechanisms. One reason that active messages perform so poorly in our implementation is the fact, that they require an additional context switch from the active message handler back to the regular user thread. Other architectures, such as the CM-5, can make active messages more efficient by providing user level access to the communication hardware.

In [5] a time of approximately $10\mu s$ is reported to write four words[3] into a remote node on a CM-5 running at 40 MHz. This has to be compared to $70\mu s$ using portals under SUNMOS on the Intel Paragon. However, it should be noted that the active message layer on the CM-5 has to poll for the data. This means that a node busy doing computations might not handle requests as quickly as the benchmarks might suggest.

[3] 4 data words and a function address = 20 bytes.

Acknowledgments

We wish to thank T. Mack Stallcup and Michael C. Proicou of Intel's Super Computing Division for their assistance with hardware issues on our Paragon machines. Thanks also go the the remaining SUNMOS/Puma team for input and assistance to this paper.

References

[1] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[2] Arthur B. Maccabe and Stephen R. Wheat. Message passing in PUMA. Technical report SAND93-0935, Sandia National Laboratories, 1993.

[3] Rolf Riesen, Arthur B. Maccabe, and Stephen R. Wheat. Split-C and active messages under SUNMOS on the Intel Paragon. Submitted to Supercomputing '94, 1994.

[4] Bernard Traversat, Bill Nitzberg, and Sam Fineberg. Experience with SUNMOS on the Paragon XP/S-15. In *Proceedings of the Intel Supercomputer User's Group. 1994 Annual North America Users' Conference.*, June 1994.

[5] Lewis W. Tucker and Alan Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4):481–496, April 1994.

[6] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM Press.

[7] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65. IEEE Computer Society Press, 1994.

DATE
FILMED
9/7/94
END